Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection [⋆]

Cristiano Calcagno[1], Walid Taha[2], Liwen Huang[3], and Xavier Leroy[4]

[1] Imperial College London, UK, ccris@doc.ic.ac.uk
[2] Rice University, Houston, TX, USA, taha@cs.rice.edu
[3] Yale University, New Haven, CT, USA, liwen.huang@yale.edu
[4] INRIA, Roquencourt, France, Xavier.Leroy@inria.fr

Abstract. The paper addresses theoretical and practical aspects of implementing multi-stage languages using abstract syntax trees (ASTs), gensym, and reflection. We present an operational account of the correctness of this approach, and report on our experience with a bytecode compiler called MetaOCaml that is based on this strategy. Current performance measurements reveal interesting characteristics of the underlying OCaml compiler, and illustrate why this strategy can be particularly useful for implementing domain-specific languages in a typed, functional setting.

1 Introduction

Program generation, partial evaluation (PE) [22], dynamic compilation, and runtime code generation (RTCG) [24] are all techniques to facilitate writing generic programs without unnecessary runtime overheads. These systems have proven effective in challenging domains including high-performance operating systems [36, 9, 8]. Multi-stage languages have been developed as a uniform, high-level, and semantically-based view of these diverse techniques. These languages provide three simple annotations to allow the programmer to distribute a computation into distinct stages [46, 40, 48].

1.1 Multi-stage Basics

Multi-stage programming languages provide a small set of constructs for the construction, combination, and execution of delayed computations. Programming in a multi-stage language such as MetaOCaml [29] can be illustrated with the following classic example[5]:

⋆ Funded by NSF ITR-0113569.

[5] Dots are used around brackets and escapes to disambiguate the syntax in the implementation. They are dropped when we talk about the underlying calculus rather than the implementation.

```
let even n = (n mod 2) = 0
let square x = x * x
let rec power n x = (* int -> .<int>. -> .<int>. *)
  if n=0 then .<1>.
   else if even n
         then .<square .~(power (n/2) x)>.
         else .<.~x * .~(power (n-1) x)>.
let power72 =                        (* int -> int *)
  .! .<fun x -> .~(power 72 .<x>.)>.
```

Ignoring the type constructor `.<`t`>.` and the three staging annotations brackets `.<`e`>.`, escapes `.~`e and run `.!`, the above code is a standard definition of a function that computes x^n, which is then used to define the specialized function x^{72}. Without staging, the last step just produces a closure that invokes the power function every time it gets a value for x. The effect of staging is best understood by starting at the end of the example. Whereas a term `fun x ->` e `x` is a value, an annotated term `.<fun x -> .~(`e `.<x>.)>.` is not. Brackets indicate that we are constructing a future stage computation, and an escape indicates that we must perform an immediate computation *while* building the bracketed computation. The application e `.<x>.` has to be performed even though x is still an uninstantiated *symbol*. In the `power` example, `power 72 .<x>.` is performed immediately, once and for all, and not repeated every time we have a new value for `x`. In the body of the definition of the `power` function, the recursive applications of `power` are also escaped to make sure that they are performed immediately. The run `.!` on the last line invokes the compiler on the generated code fragment, and incorporates the result of compilation into the runtime system.

1.2 Problem

Filinski [15, 14] presented a denotational account of the correctness of using abstract syntax trees (ASTs) and gensym to implement two-level languages. We are currently developing a multi-stage extension of OCaml [26] called MetaOCaml [29]. This setting requires previous results to be extended to include:

1. allowing multiple levels instead of two
2. allowing cross-stage persistence
3. allowing the execution of dynamically generated code (from within the language)

Without such a generalization we cannot be confident that results established with respect to the substitution semantics of multi-stage languages (most notably type safety [12, 40, 42, 43, 32, 45]) are relevant to an implementation that employs this strategy.

Compared to two-level languages, multi-stage languages are generalized to multiple levels. More significantly, multi-stage languages allow computations in any stage to refer to values from previous stages, and provide a facility for executing future-stage computations. Unfortunately, denotational models are currently available only for restricted versions of multi-stage languages [3, 15, 14, 30, 31]. [6]

1.3 Contributions

This paper presents an operational account of the correctness of implementing multi-stage languages using ASTs, gensym, and reflection. The key technical challenge in achieving this result lies in devising an appropriate notion of "decompilation" to relate the semantics of both the source (surface) and target (implementation) languages. This notion allows us to formally state that evaluating a multi-stage program can be achieved by translating it into a mostly standard single-stage language, evaluating that program, and then decompiling the result. The single-stage language is somewhat non-standard because it supports an eval-like construct (modeling runtime compilation and execution of code) that itself refers to the compilation strategy that we have set out to verify.

The correctness result shows that with the implementation strategy presented here any free variables contained in dynamically executed code are handled in a manner consistent with the substitution semantics for multi-stage languages. The result also justifies various subtle choices in the translation, such as why runtime calls to the compiler are passed the empty environment, and exactly when freshly generated names can be safely viewed as variables. By separating the correspondence between variables and names from the other concerns we were able to easily formalize and prove correctness of execution of open code.

Although the formal development is carried out with a minimal calculus, the strategy has been used to implement a multi-stage extension of the OCaml language (called MetaOCaml). The paper reports on our experience so far with this implementation. Current findings indicate that gains from staging can be less in some typed functional settings than when partially evaluating other programming languages such as C. At the same time, significant gains (in the typed functional setting) are possible when the implementation strategy studied here is used in conjunction with runtime source-to-source transformations such as tag elimination.

1.4 Organization of this paper

Section 2 introduces the syntax and semantics of a multi-stage calculus. Section 3 defines a minimal single-stage language with minimal support for ASTs, a *weak* gensym operation, and reflection. Section 4 presents the translation from

[6] Reflective towers often come with a denotational model [37, 38, 10, 27]. But a reflective tower is a sequence of one language interpreting the next, and are quite different from multi-stage languages in terms of formal meta-theory. Specifically, core issues of interest to this paper, such as evaluation under lambda, have not been addressed in studies on reflective towers.

$$\frac{}{\alpha \overset{n}{\hookrightarrow} \alpha} \qquad \frac{}{\lambda x.e \overset{0}{\hookrightarrow} \lambda x.e} \qquad \frac{e_1 \overset{0}{\hookrightarrow} \lambda x.e \quad e_2 \overset{0}{\hookrightarrow} v_1 \quad e[x := v_1] \overset{0}{\hookrightarrow} v_2}{e_1\ e_2 \overset{0}{\hookrightarrow} v_2}$$

$$\frac{e \overset{0}{\hookrightarrow} \langle v_1 \rangle \quad v_1 \overset{0}{\hookrightarrow} v_2}{!\,e \overset{0}{\hookrightarrow} v_2} \qquad \frac{}{x \overset{n+1}{\hookrightarrow} x} \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\lambda x.e \overset{n+1}{\hookrightarrow} \lambda x.v}$$

$$\frac{e_1 \overset{n+1}{\hookrightarrow} v_1 \quad e_2 \overset{n+1}{\hookrightarrow} v_2}{e_1\ e_2 \overset{n+1}{\hookrightarrow} v_1\ v_2} \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\langle e \rangle \overset{n}{\hookrightarrow} \langle v \rangle} \qquad \frac{e \overset{0}{\hookrightarrow} \langle v \rangle}{{\sim}e \overset{1}{\hookrightarrow} v} \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{{\sim}e \overset{n+2}{\hookrightarrow} {\sim}v} \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{!\,e \overset{n+1}{\hookrightarrow} !\,v}$$

Fig. 1. Big-Step Semantics for CBV λ-U

the multi-stage language to the minimal single stage language, as well as the de-compilation strategy. Section 5 develops the correctness result for the translation. Section 6 discusses practical aspects of using this strategy to implement a multi-stage extension of the OCaml language, and reports on our experience with this effort. Section 7 reviews related work, and Section 8 concludes.

2 A Multi-stage Source Language

The formal development is based on a variant of the λ-U calculus [40, 42] as the source language. The syntax for this variant is as follows:

$$e \in E ::= \alpha \mid x \mid e\,e \mid \lambda x.e \mid \langle e \rangle \mid {\sim}e \mid !\,e$$

Terms $e \in E$ are built using two disjoint sets: variables $x, y \in \mathcal{V}$ and constants $\alpha \in \mathcal{A}$. The constants α are an addition to the standard λ-U. They are included because they make defining a de-compilation function convenient. In the target language, these constants will be used to represent dynamically generated names. We write $FV(e)$ for the set of variables occurring free in e, and $FA(e)$ for the set of names occurring in e.

To ensure that we work only with terms where there is a sensible correspondence between escapes and brackets, the development does not refer directly to the set E, but rather, to the following families of expressions E^n and values V^n:

$$\begin{array}{lll}
e^0 & \in E^0 & ::= \alpha \mid x \mid \lambda x.e^0 \mid e^0\ e^0 \mid \langle e^1 \rangle \mid !\,e^0 \\
e^{n+1} & \in E^{n+1} & ::= \alpha \mid x \mid \lambda x.e^{n+1} \mid e^{n+1}\ e^{n+1} \mid \langle e^{n+2} \rangle \mid {\sim}e^n \mid !\,e^{n+1} \\
v^0 & \in V^0 & ::= \alpha \mid \lambda x.e^0 \mid \langle v^1 \rangle \\
v^{n+1} & \in V^{n+1} & = E^n
\end{array}$$

The call-by-value big-step semantics for the language is presented in Figure 1.

Lemma 1. *$e \in E^n$ and $e \overset{n}{\hookrightarrow} v$ implies $v \in V^n$.*

$$A, \lambda x.e' \hookrightarrow \lambda x.e' \qquad \frac{A, e'_1 \hookrightarrow \lambda x.e' \quad A, e'_2 \hookrightarrow v'_1 \quad A, e'[x := v'_1] \hookrightarrow v'_2}{A, e'_1\ e'_2 \hookrightarrow v'_2}$$

$$\frac{}{A, \alpha \hookrightarrow \alpha} \qquad \frac{\alpha \notin A \quad A\alpha, e'[x := \alpha] \hookrightarrow v'}{A, \mathsf{let}\ x = \mathsf{gensym}\ ()\ \mathsf{in}\ \mathsf{Lam}\ (x, e') \hookrightarrow \mathsf{Lam}\ (\alpha, v')} \qquad \frac{A, e' \hookrightarrow v'}{A, c\ e' \hookrightarrow c\ v'}$$

$$\frac{\begin{array}{c} A, e'_1 \hookrightarrow v'_1 \\ A, e'_2 \hookrightarrow v'_2 \end{array}}{A, c\ (e'_1, e'_2) \hookrightarrow c\ (v'_1, v'_2)} \qquad \frac{A, e' \hookrightarrow c\ v'}{A, \mathsf{un}c\ e' \hookrightarrow v'} \qquad \frac{A, e' \hookrightarrow \mathsf{Mrk}\ v'_1 \quad A, \left[\!\left[|v'_1|^1_\emptyset \right]\!\right]^0_\emptyset \hookrightarrow v'_2}{A, \mathsf{mor}\ e' \hookrightarrow v'_2}$$

Fig. 2. Semantics of target language

3 A Single-stage Target Language

The target language has the following syntax:

$$\begin{aligned} e' \in E' ::= \alpha \mid x \mid \lambda x.e' \mid e'e' \mid \mathsf{let}\ x{=}\mathsf{gensym}()\ \mathsf{in}\ \mathsf{Lam}\ (x, e') \\ \mid c\ e' \mid c\ (e', e') \mid \mathsf{un}c\ e' \mid \mathsf{mor}\ e' \end{aligned}$$

It contains lambda terms, names α, a gensym construct, unary and binary AST constructors, de-constructors and a run construct. A term $c\ e'$ is an application of a value constructor (or attachment of a "union type" tag) c to the result of the expression e'. The essential feature of a de-constructor is that $\mathsf{un}c(c\ e)$ evaluates to the result of e. For example, if Mrk is a constructor then unMrk is the corresponding de-constructor. The MetaOCaml Run (mor) construct represents the action of the overall implementation of MetaOCaml on a program.

Staging constructs are translated into operations on ASTs. The constructors are as follows:

$$c \in C ::= \mathsf{Var} \mid \mathsf{Lam} \mid \mathsf{App} \mid \mathsf{Brk} \mid \mathsf{Esc} \mid \mathsf{Run} \mid \mathsf{Csp} \mid \mathsf{Mrk}$$

Intuitively, the constructors will mimic an ML datatype:

```
type exp = Var of const | App of exp * exp | Lam of const * exp | Brk of exp
           | Esc of exp | Run of exp | Csp of exp | Mrk of exp
```

where each variant (respectively) represents a production in the syntax E of the *source* language. The first three are for standard lambda terms. Note that the value carried by the Var and the first value carried by Lam construct will be a constant α representing a name. The next three capture the rest of the syntax in the source language. Csp and Mrk are technical devices that do not occur in the source language, but which will be explained in more detail in the next section.

The set of values that can result from evaluation is defined as follows:

$$v' \in V' ::= \lambda x.e' \mid c\ v' \mid c\ (v', v') \mid \alpha$$

The big-step semantics for the target language is presented in Figure 2. The semantics is given by a judgment $A, e' \hookrightarrow v'$ where A is a finite set of names.

$$
\begin{aligned}
&\boxed{[\bullet]^n_\tau : E \to E'} \\
&[x]^0_{\tau,x^t} &&= x \\
&[\lambda x.e]^0_\tau &&= \lambda x.\,[e]^0_{\tau,x^0} \\
&[e_1\ e_2]^0_\tau &&= [e_1]^0_\tau\ [e_2]^0_\tau \\
&[\langle e\rangle]^0_\tau &&= \mathsf{Mrk}\ [e]^1_\tau \\
&[!\,e]^0_\tau &&= \mathsf{mor}\ [e]^0_\tau \\
&[\alpha]^0_\tau &&= \alpha \\
\\
&[x]^{n+1}_{\tau,x^+} &&= \mathsf{Var}\ x \\
&[x]^{n+1}_{\tau,x^0} &&= \mathsf{Csp}\ x \\
\\
&[\lambda x.e]^{n+1}_\tau &&= \begin{cases}\mathsf{let}\ x = \mathsf{gensym}\ () \\ \mathsf{in\ Lam}\ (x, [e]^{n+1}_{\tau,x^+})\end{cases} \\
\\
&[e_1\ e_2]^{n+1}_\tau &&= \mathsf{App}\ ([e_1]^{n+1}_\tau, [e_2]^{n+1}_\tau) \\
&[\langle e\rangle]^{n+1}_\tau &&= \mathsf{Brk}\ [e]^{n+2}_\tau \\
&[\tilde{}\,e]^1_\tau &&= \mathsf{unMrk}\ [e]^0_\tau \\
&[\tilde{}\,e]^{n+2}_\tau &&= \mathsf{Esc}\ [e]^{n+1}_\tau \\
&[!\,e]^{n+1}_\tau &&= \mathsf{Run}\ [e]^{n+1}_\tau \\
&[\alpha]^{n+1}_\tau &&= \mathsf{Var}\ \alpha
\end{aligned}
$$

$$
\begin{aligned}
&\boxed{|\bullet|^n_\tau : E' \to E} \\
&|x|^0_{\tau,x^t} &&= x \\
&|\lambda x.e'|^0_\tau &&= \lambda x.\,|e'|^0_{\tau,x^0} \\
&|e'_1\ e'_2|^0_\tau &&= |e'_1|^0_\tau\ |e'_2|^0_\tau \\
&|\mathsf{Mrk}\ e'|^0_\tau &&= \langle\ |e'|^1_\tau\ \rangle \\
&|\mathsf{mor}\ e'|^0_\tau &&= !\ |e'|^0_\tau \\
&|\alpha|^0_\tau &&= \alpha \\
\\
&|\mathsf{Var}\ x|^{n+1}_{\tau,x^+} &&= x \\
&|\mathsf{Csp}\ x|^{n+1}_{\tau,x^0} &&= x \\
&|\mathsf{Csp}\ v'|^{n+1}_\tau &&= |v'|^0_\tau \\
&\left|\begin{array}{l}\mathsf{let}\ x = \mathsf{gensym}\ () \\ \mathsf{in\ Lam}\ (x, e')\end{array}\right|^{n+1}_\tau &&= \lambda x.\,|e'|^{n+1}_{\tau,x^+} \\
&|\mathsf{Lam}\ (\alpha, v')|^{n+1}_\tau &&= \lambda x.(|v'|^{n+1}_\emptyset\ [\alpha := x]) \\
&|\mathsf{App}\ (e'_1, e'_2)|^{n+1}_\tau &&= |e'_1|^{n+1}_\tau\ |e'_2|^{n+1}_\tau \\
&|\mathsf{Brk}\ e'|^{n+1}_\tau &&= \langle\ |e'|^{n+2}_\tau\ \rangle \\
&|\mathsf{unMrk}\ e'|^1_\tau &&= \tilde{}\ |e'|^0_\tau \\
&|\mathsf{Esc}\ e'|^{n+2}_\tau &&= \tilde{}\ |e'|^{n+1}_\tau \\
&|\mathsf{Run}\ e'|^{n+1}_\tau &&= !\ |e'|^{n+1}_\tau \\
&|\mathsf{Var}\ \ \alpha|^{n+1}_\tau &&= \alpha
\end{aligned}
$$

Fig. 3. Translation and De-compilation Functions

We have chosen to use a "weak gensym" instead of one where a name-state A is threaded through the semantics. Our semantics of gensym picks a name non-deterministically. It is weak in the sense that it does not restrict the set of names that can be picked as much as the name-state gensym would. It is nevertheless sufficient to prove the correctness result presented here. This choice both simplifies presentation and provides a slightly stronger result.

The special construct mor (short for "MetaOCaml's run") is the operation that is intended to implement the more abstractly-specified construct run. Because defining the semantics of this construct itself depends on the definition of the translation, it is explained in Section 4.

4 Translation from Source to Target

This section presents the definition for both the translation and de-compilation functions. Whereas the translation is used to implement multi-stage languages using a single-stage language, the de-compilation function is purely a mathematical construction used to prove the soundness of the translation.

To define the translation, we need to associate variables to times and instantiate names with variables:

Times	$t \in T ::= 0 \mid +$
Binding times	$\tau \in B ::= \emptyset \mid \tau, x^t$
Instantiations	$\rho \in R ::= [\alpha_i := x_i]$

Notation 1 *Instantiations are substitutions where any variables or names (be it an "x" or an "α") occurring anywhere in them must be distinct: they form a one-to-one correspondence between variables and names.*

Definition 1. *The application of an instantiation to a source language expression, written $e[\alpha_i := x_i]$ denotes the result of performing the multiple substitutions (without capturing any x_i).*

Note that bound variables are not captured by substitutions and instantiations. However, there are no binders for names, so for instance $\mathsf{Lam}\ (\alpha, x)[x := \alpha] = \mathsf{Lam}\ (\alpha, \alpha)$.

The left hand side of Figure 3 defines the **translation** function $[\![\bullet]\!]^n_\tau : E \to E'$, where n is a natural number called the *level* of the term, and τ is an environment mapping free variables to binding times. Intuitively, the level of the term will just be the number of surrounding brackets less the number of surrounding escapes. The result of the translation consists mostly of either unchanged programs, operations to build ASTs (using constructors like Var, Lam, etc), or the use of the two functions gensym and mor. At level 0, the translation $[\![\bullet]\!]^0_\tau$ does essentially nothing but traverse the term. At levels $n+1$, for the most part, the translation takes a term and generates a new term that represents that first one.

There are two cases for variables at level $n+1$, depending on whether the variable was bound at level 0 or any other level. If the variable was bound at level zero, then we construct a marker around it as a cross-stage persistent (CSP) constant, otherwise it is treated as expected. The next interesting case is that of a lambda abstraction, where the essential trick is to use a fresh-name function to generate a new name so as to avoid accidental name capture. Such a function is used in the translation by Gomard and Jones [20] and before that in hygienic macro systems [25, 13]. This trick is used only in the implementation and it is completely invisible to the user. Relieving the programmer from having to explicitly ensure that accidental capture is avoided is an important feature of multi-stage languages, from both the programmer's and the language designer's point of view (c.f. [16] for an example of the latter).

From the practical point of view, an important feature of the translation is that it does not need a substitution function (which is costly) or a complex environment. The key idea is to use the same name for variables that are used in the input in the output. The types of these variables certainly change (in the input the type can vary, but in the output it is always of "identifier"). Care must be taken, however, to make sure that the scoping of these variables in the original program is not accidentally modified (we give an example of this in section 6.1).

For example, if the translation encounters the term $(\lambda x.x)\ (\lambda y.y)$ at level 1, it generates the term

$$\mathsf{App}(\mathsf{let}\ x = \mathsf{gensym}()\ \mathsf{in}\ \mathsf{Lam}(x, \mathsf{Var}\, x), \mathsf{let}\ y = \mathsf{gensym}()\ \mathsf{in}\ \mathsf{Lam}(y, \mathsf{Var}\, y))$$

At level $n+1$ the source-to-source translation roughly generates the AST for an expression which will construct at runtime the AST for the specialized program that will be passed to the runtime compiler and evaluator (mor). For Brackets and Escapes, we always adjust the level parameter. There is no case for Esc at level 0 (such terms are rejected by the expression families). When the translation finds a Bracket, the level is raised by one. Note that at the lowest levels the translation for Brackets generates Mrk and the translation for escape generates unMrk. While it may seem unnecessary to do that, it simplifies the formal treatment of this translation. Without Mrk and unMrk, terms such as $\tilde{\ }\langle e\rangle$, for example, would have the same translation as e. If this were the case, then the translation would have no functional inverse. Having such a functional inverse (de-compilation) is instrumental for the formal development presented here.

At level 0, Run is replaced by an application of the constant mor. Recall the formal semantics of mor in Figure 2. The de-compilation $|\bullet|^1_\emptyset$ corresponds to passing to the compiler a value representing source code, since the ASTs used in the implementation are those used by the compiler to represent ASTs. However, this de-compilation produces a source language term that contains constructs such as $\langle\ \bullet\ \rangle$ and $\tilde{\ }\bullet$ that the target language does not understand. Thus, it is necessary to apply the very translation that we are presenting here once again: $[\![\bullet]\!]^0_\emptyset$. It is useful to note that, in the big-step semantics for Run, de-compilation treats the term as a level 1 term, but translation treats it as a level 0 term. This is justified by the fact that, in the source language, $E^n = V^{n+1}$.

The right hand side of Figure 3 defines the **de-compilation** function. For all the forms *produced* by the translation, the de-compilation function simply tries to invert the translation by producing something of the same form as the translation using the de-compilation of the sub-terms. The interesting cases in the de-compilation function are the ones that take as input forms not directly generated by the translation, but which arise during the evaluation of a term produced by the translation. In fact, in what follows we show that the de-compilation function deals with *all* the new forms generated during evaluation in the target. There are only two new forms that arise: one for cross-stage persistent (CSP) variables at level $n+1$, and abstractions at level $n+1$. For CSP variables, we need a more general form which is closed under substitution. Also, we know that these variables are only ever substituted by level 0 values, so, we decompile these values at that level. Level $n+1$ abstractions were the most subtle to deal with and are probably the most interesting case in the de-compilation function. Once the form generated by the translation (let x =gensym()...) is evaluated, it causes all the occurrences of the variable x to be replaced by a fresh name α. This is the only case where instantiation $[\alpha := x]$ is used. Thus, it is only at that point that a dynamically generated name is viewed as a variable name. The fact that the translation of the sub-term in this case is carried out in the empty environment will be essential for the correctness proof. The definition is justified by the observation that the target semantics produces new names but no new free variables during evaluation (and that evaluation begins with a closed term).

Whereas translation is total, de-compilation is partial. The following definition will be convenient for denoting the domain of the de-compilation function:

Definition 2. *We write $\tau \overset{n}{\vdash} e'$ when $|e'|^n_\tau$ is defined.*

Lemma 2 (Basic Properties). *$\tau \overset{n}{\vdash} e'$ implies that*

1. *$FV(|e'|^n_\tau) \subseteq dom(\tau)$, and*
2. *$|e'|^n_\tau \in E^n$, and*
3. *$e' \in V'$ implies $|e'|^n_\tau \in V^n$.*

5 Correctness

The compilation function does not introduce any information loss:

Lemma 3 (Inversion). *$e \in E^n$ and $FV(e) \subseteq dom(\tau)$ implies $e = |[\![e]\!]^n_\tau|^n_\tau$.*

Thus $|e'|^n_\tau$ is surjective, which makes it possible (in the main theorem) to quantify over all n, e', and τ to cover the set of all source programs E.

The above property is the only one that we establish about translation. In the rest of the development, all the work goes into establishing properties of the de-compilation function.

Lemma 4 (Weakening). *$\tau \overset{n}{\vdash} e'$ and $x \notin \tau$ implies $|e'|^n_\tau = |e'|^n_{\tau,x^t}$.*

Lemma 5 (Substitution). *There are two distinct cases:*

1. *$\tau, x^0 \overset{n}{\vdash} e'$ and $\tau \overset{0}{\vdash} v'$ and $v' \in V'$ implies*
$$|e'|^n_{\tau,x^0}\left[x := |v'|^0_\tau\right] = |e'\left[x := v'\right]|^n_\tau$$
2. *$\tau, x^+ \overset{n}{\vdash} e'$ implies*
$$|e'|^n_{\tau,x^+}\left[x := \alpha\right] = |e'[x := \alpha]|^n_\tau$$

Proof. Both cases are by induction on the structure of e'. The most interesting case for part 2 is $\mathsf{Lam}\ (\alpha, e')$, where there is a possibility of introducing a name capture: this does not happen because de-compilation uses $|e'|_{\emptyset}^{n+1}$ which implies $FV(e') = \emptyset$ so that substitution has no effect. (This is also why the first part of the lemma holds for Lam .) $\square$

Two examples illustrate why we need two separate cases. First, $|\mathsf{Var}\ (x)|_{x+}^{1}$ is defined but not $|\mathsf{Var}\ (\lambda y.y)|_{\emptyset}^{1}$. Second, $|\mathsf{Lam}\ (\alpha, x)[x := \alpha]|_{\emptyset}^{1}$ is defined but $|\mathsf{Lam}\ (\alpha, x)|_{x+}^{1}$ is not.

The key observation about names is the following:

Lemma 6 (Subject Reduction w.r.t. $FA(|-|_{\emptyset}^{n})$**).** *For all* $\emptyset \overset{n}{\vdash} e'$, *whenever* $A, e' \hookrightarrow v'$ *then* $\emptyset \overset{n}{\vdash} v'$ *and* $FA(|v'|_{\emptyset}^{n}) \subseteq FA(|e'|_{\emptyset}^{n})$.

In particular, names are generated dynamically in the target language, so, we certainly cannot say that the result of evaluation has no more names than the input to evaluation. What we can say, however, is that viewed as source language terms, the result of evaluation contains no more names than the term that evaluation starts of with. Having a formal notion of de-compilation is what allows us to make this statement formally.

Theorem 1 (Main). *For all* $\emptyset \overset{n}{\vdash} e'$, *whenever* $|e'|_{\emptyset}^{n}\ \rho \overset{n}{\hookrightarrow} v$ *and* $dom(\rho) \cup FA(|e'|_{\emptyset}^{n}) \subseteq A$, *there exists* v' *such that* $A, e' \hookrightarrow v'$ *and* $|v'|_{\emptyset}^{n}\ \rho = v$.

Proof. By case analysis on e' and n, and by induction on the derivation of $|e'|_{\emptyset}^{n}\ \rho \overset{n}{\hookrightarrow} v$.

From inversion and the main theorem it is easy to see that

Corollary 1 (Simulation). *If* $e \in E^{n}$ *and* $FV(e) = \emptyset$ *and* $FA(e) \subseteq A$ *and* $e \overset{n}{\hookrightarrow} v$, *then there exists* v' *such that* $A, [\![e]\!]_{\emptyset}^{n} \hookrightarrow v'$ *and* $|v'|_{\emptyset}^{n} = v$.

6 Implementation

This section gives a more detailed description of the implementation, reports measurements on performance gains from staging and tag elimination on small example programs, and discusses expectations for performance in the native code compiled setting.

6.1 Overview

The MetaOCaml implementation is currently a modified OCaml bytecode compiler.[7] Minor changes are made to the parser and type-inference/check to accommodate the three staging annotations and a parametric type constructor

[7] OCaml also has a native code compiler, and modifying the native code compiler is still work in progress.

(to distinguish code values from regular values). Because these changes are very localized and are done by direct analogy with other constructs in the language, error reporting continues to be reliable and accurate (good error reporting is particularly hard in typed functional languages based on Hindley-Milner inference).

The bulk of the work in developing MetaOCaml is in the addition of a source-to-source translation phase right after type-inference/checking. Because we use a functional language with pattern matching to implement the translation, the implementation follows closely the formal definition presented above. The ASTs are the ones used internally by the compiler. In addition to sheer size of the AST datatype for the full language, the implementation is complicated by the fact that OCaml propagates source code file information in each construct, for very accurate error messages. In addition, the type of the AST changes during type-checking (from an AST without explicit typing annotations to a different AST with typing information), raising the need to work with two different representations. Finally, constructing ASTs that represent other ASTs is not only tedious, but type systems of languages like ML (which do not provide the necessary dependent typing structure) do not help in this regard. This problem has been a significant practical challenge for our work.

Binding Constructs In principle, once we have addressed lambda at levels higher than 1 correctly, we have captured the essence of how all binding constructs should be handled. In practice, some care is still needed when dealing with syntactic sugar for other binding constructs. For example, it is tempting to translate let as follows:

$$[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]^{n+1}_{\tau} = \begin{cases} \mathsf{let}\, x = \mathsf{gensym}\ () \\ \mathsf{in}\ \mathsf{Let}\ (x, [\![e_1]\!]^{n+1}_{\tau}, [\![e_2]\!]^{n+1}_{\tau;x\mapsto(n+1)}) \end{cases}$$

Whereas an x in e_1 would have been bound somewhere completely outside this expression, in the new term, we are replacing it every time by a completely new fresh name that is not bound in the scope of e_2. The correct translation is as follows:

$$[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]^{n+1}_{\tau} = \begin{cases} \mathsf{let}\, z = [\![e_1]\!]^{n+1}_{\tau} \\ \quad x = \mathsf{gensym}\ () \\ \mathsf{in}\ \mathsf{Let}\ (x, z, [\![e_2]\!]^{n+1}_{\tau;x\mapsto(n+1)}) \end{cases}$$

Datatypes and Exceptions In OCaml, datatype constructors as well as case-statements are partially interpreted (or "compiled") during the type-checking phase. The reason is that runtime values of datatypes do not contain constructor names, but instead integer tags[8] identifying the constructors, and determined during type-checking by examination of the datatype declaration to which the

[8] Note that the use of "tag" here and in "tag elimination" are (again) different. Here a tag is an integer, there a tag is a constructor.

constructors belong. Thus it would be incorrect to build a code fragment that uses just constructor names. If this is done, and the code is type-checked and executed in the context of another declaration for another datatype that happens to use the same constructor name, the representation of that constructor would be determined in the wrong typing environment (dynamic scoping), and type safety would also be violated. This issue would arise with the program:

```
type t = C of int;; let v = ⟨C 7⟩;; type t = C of string;; ! v;;
```

The correct implementation would allow us to execute the last statement safely. To deal with this problem, we interpret all constructors and case-statements when we type-check the source program for the first time (in this case, our implementation is doing a simple kind of staged compilation). In addition, the AST datatype is extended for these two cases with an extra field that tells us if these constructs have been interpreted before or not. The first time they are type-checked, their interpretation is recorded. If they are type-checked again, they are not re-interpreted, but rather, their original interpretation is used directly. This treatment parallels the implementation of CSP which allows us to incorporate pointers to previously compiled (and even executed) values in ASTs.

We expected exceptions to be similar to datatypes. But because there is only one global name space for exceptions, and they are represented at runtime by their names, the treatment of exceptions is in fact simpler than for datatypes: all that needs to be done is to construct a code fragment that carries the name of the exception.

Cross-Stage Persistence (CSP) Cross-stage persistence on ground values is known as lifting [22]. For literals, a simple optimization on the basic scheme is quite desirable, namely, simply producing an AST that would later just compile to that literal. Doing this has two advantages: First, a pointer de-reference is eliminated, making such constants a bit more efficient. Second, it enables further optimizations such as constant propagation and strength reduction, which would otherwise be blocked.

Run In the implementation, assuming that native-run is the existing compilation routine, mor is defined as:

$$\mathsf{mor\ p} = \mathsf{native\text{-}run\ env0}\ (\llbracket \mathsf{p} \rrbracket^0_\emptyset)$$

Where env0 is the empty environment except for pervasive constants. The compiler (composed with a jump to the compiled code) denoted here by native-run corresponds to the evaluation relation $\hookrightarrow$.

6.2 Performance Gains from Staging

One of the key goals behind developing MetaOCaml is to collect concrete data about multi-stage programming. At this point, we have only collected preliminary data on a number of small examples. Larger and more extensive studies are

Name	**Run**	**Generate**	**Compile**	**Run (2)**	**Factor**	**BEP**
power	$(7.44s/10^6)$=1x	2.65x	336x	0.18x	5.43	416
dot	$(2.14s/10^5)$=1x	39.4x	3490x	0.80x	1.25	17500
dotU	$(1.68s/10^5)$=1x	49.1x	4360x	0.72x	1.40	15400
eval "fib 15"	$(6.48s/10^3)$=1x	0.00529x	0.348x	0.235x	4.25	1
TE(eval "fib 15")	$(6.48s/10^3)$=1x	0.00338x	0.348x	0.0225x	44.4	1
rewrite	$(6.61s/10^6)$=1x	13.2x	1200x	0.90x	1.11	12100
rewriteCPS	$(7.42s/10^6)$=1x	5.69x	229x	0.08x	13.1	255
chebyshev	$(1.37s/10^4)$=1x	8.03x	1010x	0.32x	3.08	1510
chebyshevU	$(1.37s/10^4)$=1x	8.39x	1050x	0.32x	3.12	1550

Fig. 4. Speedup Factors and Break-even Points (BEPs) with Staging

under way. In this section, we present our findings on these small examples. For the most part they are encouraging and are consistent with the experience of other researchers with PE and RTCG systems. The data points to the utility of multi-stage languages for building simple yet efficient DSLs, and also highlights some peculiarities of both the OCaml bytecode setting.

What is Measured Figure 4 tabulates the timings for a set of small benchmarks, and two metrics based on the raw data. The first column, **Run**, contains a pair (t/n) where t is the total time in seconds for n runs of an unstaged version of the program at hand. The number of n is chosen so as to 1) be at least 10, and 2) so that the total time is at least 0.5 seconds. [9] The rest of the timings are for a staged version of the program, but are all normalized with respect to the time of the unstaged program. **Generate** is the time needed to perform the first stage of the staged program, which involves the generation of a code fragment. **Compile** is the time needed to compile that fragment using the .! construct. **Run (2)** is the time needed to run the code that results from compiling that fragment. The next two columns are computed from the first four. **Factor** is the first run time divided by the second run time. This is an indicator of the improvement when generation and compilation times can be amortized. **Break-even point (BEP)** is the number of times the program must be run before the total time of running the staged version (including a one-time generation and compilation cost) would be greater than the time of running the unstaged version. This is an alternative measure of the effectiveness of staging.

[9] Measurements collected on a P-III 900M machine with 128MB of main memory running Red Hat Linux 6.1 using the MetaOCaml timing function found in the Trx module. The benchmarks instrumented with the measuring functions are part of the distribution in the mex/benchmark1/ directory. The MetaOCaml_302_alpha_002 distribution is used, and is available online [29].

6.3 Experiments

We have considered the following programs:

• power: The example described in Section 1. The staged version runs about 5.4 times as fast as the unstaged program. The gain of performance comes from moving the recursion and function calls from runtime to code generation time.

• dot: Dot product turns out to be a poor example for staging, not because of the small performance gains, but rather the very large compilation times. Compilation times are large because the code generated by the first stage is essentially a long sequence of multiplication and addition expressions (linear in the size of the array). This situation does not improve significantly even with RTCG [34].

Nevertheless, this example points out the issue of how many garbage-collected languages deal with arrays, because our gains were slightly lower than those reported in Tempo [34]. In the setting of Tempo, when the array and the index are both known at RTCG time, the computation of the address of the array element can be done entirely, resulting in a constant address being put in the generated code. For garbage-collected languages such as OCaml, arrays are dynamically managed by the garbage collector, which might even relocate them, so it is not possible to treat the array pointer as a constant. It is still possible to exploit the knowledge of the array index and the array size to 1) simplify the address computation, and 2) eliminate the runtime bound check (like Java, OCaml enforces array bounds checking). The OCaml native-code compiler implements some of these optimizations, but the bytecode compiler that MetaOCaml uses does not. Because of the overhead involved in interpreting bytecode, in our setting, the specialization of array accesses does not improve performance significantly. Switching off bounds checking in our experiment does not improve runtimes significantly.

• eval "fib 15": The eval function is an interpreter for a lambda calculus with recursion, arithmetic, and conditionals. It is a particularly good example of staging. This can be seen both by the relatively high speedup, but more importantly, by the optimal break-even point.

• TE(eval "fib 15"): This is exactly the same example as above, but with tag elimination [44] performed on the result of the first stage. The speed up is substantially higher. Note also that compilation times are lower, because tag elimination generally reduces the size of the program. Tag elimination is discussed in the next section.

• rewrite: This is a matching function that takes a left-hand side of a rule and a term and returns either the same term or the corresponding right-hand side [40, 48]. Staging this program directly produces mediocre results, but staging it after it is CPS-converted produces significantly better results. Rewriting a program into CPS is one of many "binding-time improvements" that can make programs more amenable to staging [22].

• chebyshef: This is an example from a study on specializing scientific programs by Glück *et al.* [19]. Although we achieve some speedup, our results are weaker

eval "..."	n	**R**	**R1**	**R2**	**R/R1**	**R1/R2**	**R/R2**
arithmetic	10^7	19.4s	10.4 s	3.20s	1.87	3.25	6.06
fact 0	10^7	23.8s	5.62s	3.51s	4.23	1.60	6.78
fact 5	10^6	22.9s	4.19s	0.87s	5.47	4.82	26.3
fact 10	10^6	43.3s	7.71s	1.32s	5.62	5.84	32.8
fib 1	10^7	29.3s	7.94s	3.93s	3.69	2.02	7.46
fib 5	10^6	47.1s	11.3 s	1.39s	4.17	8.13	33.9
fib 10	10^5	58.5s	13.9 s	1.36s	4.21	10.2	43.0
fib 15	10^4	65.9s	15.6 s	1.48s	4.22	10.5	44.5

Fig. 5. The Effect of Staging and Tag Elimination on Runtimes

on this example than those achieved by others [19, 34]. We suspect that the reason is again the issue with array lookups in OCaml. chebyshefU is without bounds checks.

Experience with Tempo on specializing small programs indicates performance gains between 1.42 to 12.17 times [34]. In this light, the speedups achieved using staging in MetaOCaml are reasonable, although we have discussed above the typed functional settings may not be the best suited for certain kinds of computations.

6.4 Performance Gains from Tag Elimination

The previous section mentioned tag elimination, and it was seen that it can produce a significant speedup when applied to the eval example. Tag elimination is a recently-proposed source-to-source transformation that can be applied to dynamically generated code to remove typed coercions (referred to as tagging and untagging operations) that are no longer needed [44]. Tag elimination solves a long-standing problem in effectively staging (or partially evaluating) interpreters written in typed functional languages [44].

Figure 5 summarizes experiments carried out to assess tag elimination. The rows in the table correspond to different inputs to the `eval` function. The term `arithmetic` is a simple arithmetic expression (with no conditionals or recursion). The functions `fact` and `fib` are the standard factorial and Fibonacci functions. The number n is the number of times each computation is repeated so that the total time for the repetitions (**R**, **R1**, and **R2**) is a value that is easy to measure accurately. The time for running unstaged eval is **R**. The time for running the code generated by the staged interpreter is **R1**. The time for running the code generated by applying tag elimination to the result of the staged interpreter is **R2**.

In the absence of iteration or recursion, the overall gain (**R/R2**) is only 6-7 times. With iteration, overall gains are (at least in this set of examples) over 20 times. The overall gains for the Fibonacci example are the highest, and a large portion of that overall gain is due to tag elimination. Previous estimates of the potential gains from tag elimination were around 2-3 times [44]. Here it seems that tag elimination has additional benefits, probably because of enabling the compiler to perform more optimizations on the code being compiled.

6.5 Bytecode vs. Native Code Compilers

Bytecode interpreters do not have the same timing characteristics as real processors. Bytecode interpreters cannot execute several instructions in parallel, and incur an interpretation overhead on each instruction (fetching, decompiling, and branching to the appropriate piece of code to perform the instruction) that accounts for approximately half of the total execution time. Thus, in a bytecode interpretation setting, most of the benefits of runtime code specialization comes from the reduction in the number of instructions executed, e.g. when eliminating conditional branches, or removing the loop overhead by total unrolling.

In contrast, as we mentioned in the case of arrays, specializing the arguments to a bytecode instruction gains little or nothing. A typical example is that of an integer multiplication by 5. A native code compiler might replace the multiplication instruction by a shift and an add. In a bytecode setting, this kind of strength reduction is not beneficial: the overhead of interpreting two instructions instead of one largely offsets the few cycles saved in the actual computations.

For these reasons, we believe that using a native code compiler instead of a bytecode compiler and interpreter would result in higher speedups (of staged code w.r.t. unstaged code), because this would take advantage of instruction removal and of argument specialization. On the other hand, runtime compilation times could be significantly higher, especially if the native code compiler performs non-trivial optimizations, resulting in higher break-even points. The effect of moving to the native code compiler on the break-even point, however, is much less predictable. In particular, the native code compiler can be ten times slower, and code generated by that compiler can be ten times faster. If we assume that the speedup will be constant in the native code setting (which is a pessimistic approximation), then the estimated break-even point in the native code compiler jumps up by a factor of 40 times for many of the cases above. The notable exceptions, however, are the interpreter examples, where the break-even point remains unchanged at 1. For these reasons we expect that, after analyzing the performance in the native compiler setting, staging combined with tag elimination will remain a viable approach for building staged interpreters.

7 Related Work

Kohlbecker, Friedman, Felleisen, and Duba seem to have been the first to give a formal treatment of hygiene, which is what gives rise to the need for dynamic renaming. They introduce a formal renaming system and prove that in their system hygiene (defined as a HC/ME criterion) is preserved. Their system assumes that all syntactic transform functions are known before the macro-expand process commences (no macro can create macros). Clinger and Rees's [7] follow-up on this work with two improvements. First, they show that the performance of Kohlbecker's algorithm can be improved, from quadratic down to linear. Furthermore, they generalize the Kohlbecker algorithm to handle macros that can generate macros. No formal notion of correctness is given.

The notion of two-level languages originated in work of Jones on a different kind of two-level language developed primarily to model the internal workings of PE systems [20, 23], which were in turn inspired by quasi-quotes in LISP and to Quine's corners $\ulcorner\bullet\urcorner$ [40]. (Bawden[1] gives a detailed historical review of the history of quasi-quotations in LISP.) Glück and Jørgensen [17, 18] were the first to generalize the techniques of two-level languages to a multi-level setting. Over the last six years, significant efforts have been invested in the development of a theory of multi-stage computation, starting from the work of Davies on linear temporal logic [11] and Moggi on functor-category and topos-theoretic semantics [30, 31], then continuing as various studies on the semantics [43, 2, 40] and type systems for MetaML [32, 6, 4, 5].

Significant effort has also been put in studying staging interpreters. Thibault, Consel, Lawall, Marlet, and Muller [49] and Hasuhara and Yonezawa [28] study the staging (via partial evaluation) of byte-code interpreters, and deal with many technical problems that arise in this setting. These efforts investigate real-world interpreters (such as the OCaml and Java bytecode interpreters), and it would be interesting future work to reproduce these results in MetaOCaml and to see the impact of using tag elimination (which was not available at the time of these works) in that setting.

A number of extensions of C [8, 21, 39] can be viewed as two-level languages. In these languages, delayed computations are not implemented by parse trees, but rather, directly by low-level code that is dynamically constructed at runtime (one system [8] supports both strategies). In future work we will study how this alternative strategy can be used for multi-level languages.

8 Conclusions

This paper presents an operational account of the correctness of using ASTs, gensym, and reflection to implement a multi-stage calculus on top of a single-stage calculus. The paper also reports on experience with putting this strategy to work in the context of extending the OCaml language with multi-stage constructs. On the positive side, we find that using this strategy in conjunction with transformations such as tag elimination yields significant performance improvement for concise definitional interpreters written in OCaml. On the negative side, we find that for certain applications that involve numerical computation written in OCaml, the gains obtained are not as much as those achieved by staging (or partially evaluating) programs written in C.

The present work focuses on effect-free multi-stage programming. Big-step semantics exist for multi-stage programming in the imperative setting, but can be quite involved [6, 5]. Future work will focus on studying the soundness of the strategy studied here in this more challenging setting.

MetaOCaml has been used for instruction in two graduate courses on multi-stage programming (c.f. [33]). MetaOCaml distributions are publicly available online [29].

Acknowledgments: Oleg Kiselyov pointed out a technical oversight and corrected us on related work. Antoney Courtney, Stephan Ellner, Bill Harrison, Julia Lawall, Eugenio Moggi, Henrik Nilsen, Matthai Philipose, Andrew Tolmach, and Zhanyong Wan read and commented on drafts of this paper.

References

1. Bawden, A. Quasiquotation in LISP. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (San Antonio, 1999), O. Danvy, Ed., University of Aarhus, Dept. of Computer Science, pp. 88–99. Invited talk.
2. Benaissa, Z. E.-A., Moggi, E., Taha, W., and Sheard, T. Logical modalities and multi-stage programming. In *Federated Logic Conference (FLoC) Satellite Workshop on Intuitionistic Modal Logics and Applications (IMLA)* (1999).
3. Calcagno, C., and Moggi, E. Adequacy and correctness for two-level languages. (Unpublished manuscript), 1998.
4. Calcagno, C., and Moggi, E. Multi-stage imperative languages: A conservative extension result. In *[41]* (2000), pp. 92–107.
5. Calcagno, C., Moggi, E., and Sheard, T. Closed types for a safe imperative MetaML. *Journal of Functional Programming* (2003). To appear.
6. Calcagno, C., Moggi, E., and Taha, W. Closed types as a simple approach to safe imperative multi-stage programming. In *the International Colloquium on Automata, Languages, and Programming (ICALP '00)* (Geneva, 2000), vol. 1853 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 25–36.
7. Clinger, W., and Rees, J. Macros that work. In *In proceedings of the ACM Symposium on Principles of Programming Languages (POPL)* (Orlando, 1991), ACM Press, pp. 155–162.
8. Consel, C., and Noël, F. A general approach for run-time specialization and its application to C. In *In proceedings of the ACM Symposium on Principles of Programming Languages (POPL)* (St. Petersburg Beach, 1996), pp. 145–156.
9. Consel, C., Pu, C., and Walpole, J. Incremental specialization: The key to high performance, modularity, and portability in operating systems. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (New York, 1993), ACM Press, pp. 44–46.
10. Danvy, O., and Malmkjær, K. Intensions and extensions in a reflective tower. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming* (1988), ACM Press, pp. 327–341.
11. Davies, R. A temporal-logic approach to binding-time analysis. In *the Symposium on Logic in Computer Science (LICS '96)* (New Brunswick, 1996), IEEE Computer Society Press, pp. 184–195.
12. Davies, R., and Pfenning, F. A modal analysis of staged computation. In *the Symposium on Principles of Programming Languages (POPL '96)* (St. Petersburg Beach, 1996), pp. 258–270.

13. Dybvig, R. K., Hieb, R., and Bruggeman, C. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation 5*, 4 (Dec. 1992), 295–326.
14. Filinski, A. A semantic account of type-directed partial evaluation. In *Principles and Practice of Declarative Programming (PPDP)* (1999), vol. 1702 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 378–395.
15. Filinski, A. Normalization by evaluation for the computational lambda-calculus. In *Typed Lambda Calculi and Applications: 5th International Conference (TLCA)* (2001), vol. 2044 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 151–165.
16. Ganz, S., Sabry, A., and Taha, W. Macros as multi-stage computations: Type-safe, generative, binding macros in MacroML. In *the International Conference on Functional Programming (ICFP '01)* (Florence, Italy, September 2001), ACM.
17. Glück, R., and Jørgensen, J. Efficient multi-level generating extensions for program specialization. In *Programming Languages: Implementations, Logics and Programs (PLILP'95)* (1995), S. D. Swierstra and M. Hermenegildo, Eds., vol. 982 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 259–278.
18. Glück, R., and Jørgensen, J. Fast binding-time analysis for multi-level specialization. In *Perspectives of System Informatics* (1996), D. Bjørner, M. Broy, and I. V. Pottosin, Eds., vol. 1181 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 261–272.
19. Glück, R., Nakashige, R., and Zöchling, R. Binding-time analysis applied to mathematical algorithms. In *System Modelling and Optimization* (1995), J. Doležal and J. Fidler, Eds., Chapman & Hall, pp. 137–146.
20. Gomard, C. K., and Jones, N. D. A partial evaluator for untyped lambda calculus. *Journal of Functional Programming 1*, 1 (1991), 21–69.
21. Hornof, L., and Jim, T. Certifying compilation and run-time code generation. *Higher-Order and Symbolic Computation 12*, 4 (Dec. 1999), 337–375.
22. Jones, N. D., Gomard, C. K., and Sestoft, P. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.
23. Jones, N. D., Sestoft, P., and Sondergraard, H. An experiment in partial evaluation: The generation of a compiler generator. In *Rewriting Techniques and Applications*, J.-P. Jouannaud, Ed., vol. 202 of *Lecture Notes in Computer Science.* Springer-Verlag, 1985, pp. 124–140.
24. Keppel, D., Eggers, S. J., and Henry, R. R. A case for runtime code generation. Tech. Rep. 91-11-04, University of Washington, 1991.
25. Kolhlbecker, E. E. *Syntactic Extensions in the Programming Language Lisp.* PhD thesis, Indiana University, Bloomington, Indiana, 1986.
26. Leroy, X. Objective Caml, 2000. Available from http://caml.inria.fr/ocaml/.
27. Malmkjær, K. On some semantic issues in the reflective tower. In *Mathematical Foundations of Programming Semantics. (Lecture Notes in Computer Science, vol. 442)* (1989), M. Main, A. Melton, M. Mislove, and D. Schmidt, Eds., pp. 229–246.
28. Masuhara, H., and Yonezawa, A. Run-time bytecode specialization. *Lecture Notes in Computer Science 2053* (2001), 138–??
29. MetaOCaml: A compiled, type-safe multi-stage programming language. Available online from http://www.cs.rice.edu/~taha/MetaOCaml/, 2003.
30. Moggi, E. A categorical account of two-level languages. In *Mathematics Foundations of Program Semantics* (1997), Elsevier Science.
31. Moggi, E. Functor categories and two-level languages. In *Foundations of Software Science and Computation Structures (FoSSaCS)* (1998), vol. 1378 of *Lecture Notes in Computer Science*, Springer Verlag.

32. Moggi, E., Taha, W., Benaissa, Z. E.-A., and Sheard, T. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming (ESOP)* (1999), vol. 1576 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 193–207.
33. Multi-stage programming. http://www.cs.rice.edu/~taha/teaching/02F/511, 2003.
34. Noël, F., Hornof, L., Consel, C., and Lawall, J. L. Automatic, template-based run-time specialization: Implementation and experimental study. In *Proceedings of the 1998 International Conference on Computer Languages* (1998), IEEE Computer Society Press, pp. 132–142.
35. Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000,USA. Available online from ftp://cse.ogi.edu/pub/tech-reports/README.html. Last viewed August 1999.
36. Pu, C., and Walpole, J. A study of dynamic optimization techniques: Lessons and directions in kernel design. Tech. Rep. CSE-93-007, Oregon Graduate Institute, 1993. Available from [35].
37. Smith, B. C. *Reflection and Semantics in a Procedural Language*. PhD thesis, Massachusetts Institute of Technology, 1982.
38. Smith, B. C. Reflection and semantics in LISP. In *ACM Symposium on Principles of Programming Languages* (1984), pp. 23–35.
39. Smith, F., Grossman, D., Morrisett, G., Hornof, L., and Jim, T. Compiling for template-based run-time code generation. *Journal of Functional Programming* (2002). To appear.
40. Taha, W. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [35].
41. Taha, W., Ed. *Semantics, Applications, and Implementation of Program Generation* (Montréal, 2000), vol. 1924 of *Lecture Notes in Computer Science*, Springer-Verlag.
42. Taha, W. A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trivial. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Maniplation (PEPM)* (Boston, 2000), ACM Press.
43. Taha, W., Benaissa, Z.-E.-A., and Sheard, T. Multi-stage programming: Axiomatization and type-safety. In *25th International Colloquium on Automata, Languages, and Programming (ICALP)* (Aalborg, 1998), vol. 1443 of *Lecture Notes in Computer Science*, pp. 918–929.
44. Taha, W., Makholm, H., and Hughes, J. Tag elimination and Jones-optimality. In *Programs as Data Objects* (2001), O. Danvy and A. Filinksi, Eds., vol. 2053 of *Lecture Notes in Computer Science*, pp. 257–275.
45. Taha, W., and Nielsen, M. F. Environment classifiers. In *The Symposium on Principles of Programming Languages (POPL '03)* (New Orleans, 2003).
46. Taha, W., and Sheard, T. Multi-stage programming with explicit annotations. In *Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM)* (Amsterdam, 1997), ACM Press, pp. 203–217.
47. Taha, W., and Sheard, T. MetaML and multi-stage programming with explicit annotations. Tech. Rep. CSE-99-007, Department of Computer Science, Oregon Graduate Institute, 1999. Extended version of [46]. Available from [35].
48. Taha, W., and Sheard, T. MetaML: Multi-stage programming with explicit annotations. *Theoretical Computer Science 248*, 1-2 (2000). Revision of [47].
49. Thibault, S., Consel, C., Lawall, J. L., Marlet, R., and Muller, G. Static and dynamic program compilation by interpreter specialization. *Higher-Order and Symbolic Computation 13*, 3 (Sept. 2000), 161–178.